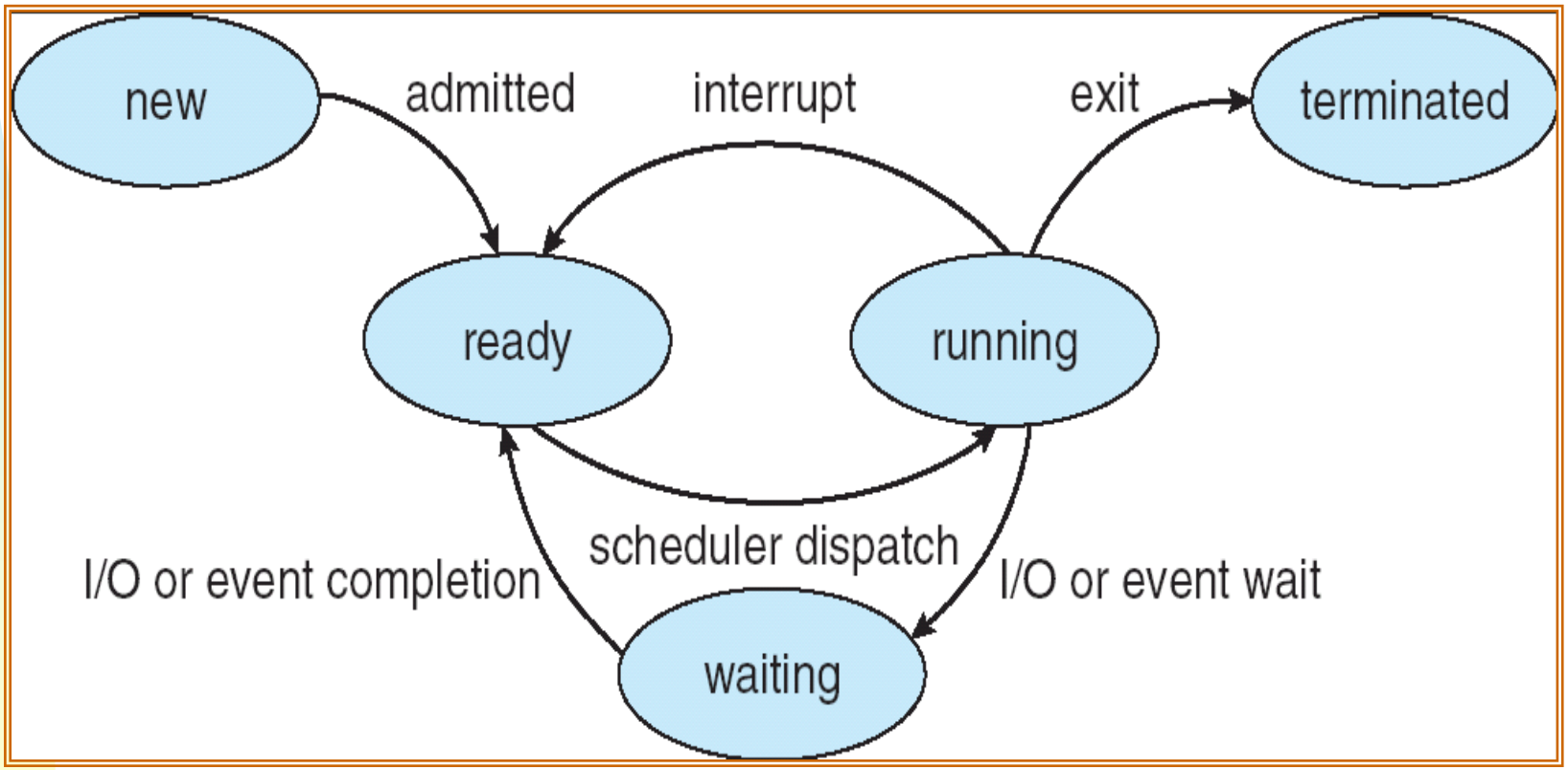
Module 4 - Process scheduling

Reading: Chapter 6 (Silberschatz)

Module overview

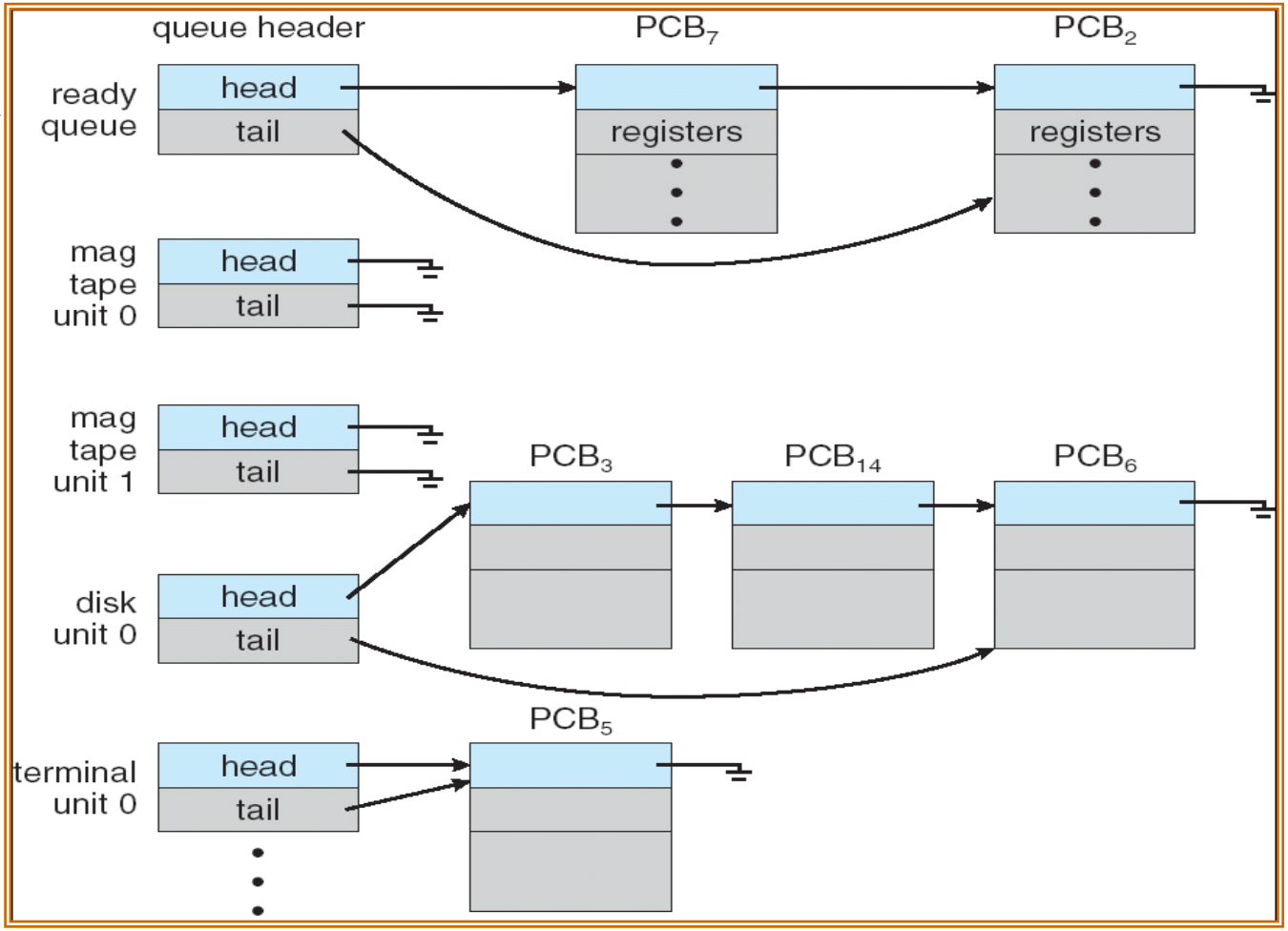
- Basic concepts
- Scheduling criteria
- Scheduling algorithms
- Multiprocessor scheduling
- Algorithm evaluation

State transition diagram of a process



Process queues for scheduling

Ready queue

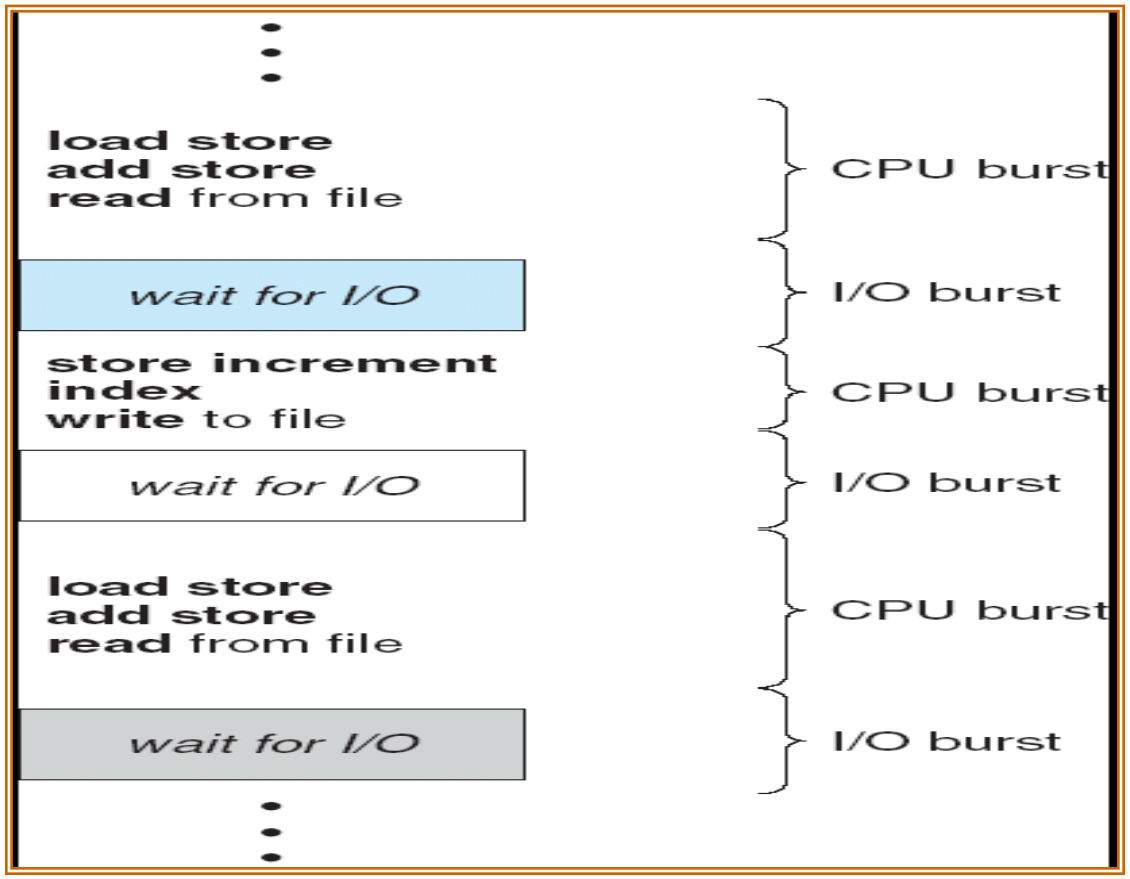


We will assume that the first process in a queue is the one that uses the resource: here, proc7 executes

Basic concepts

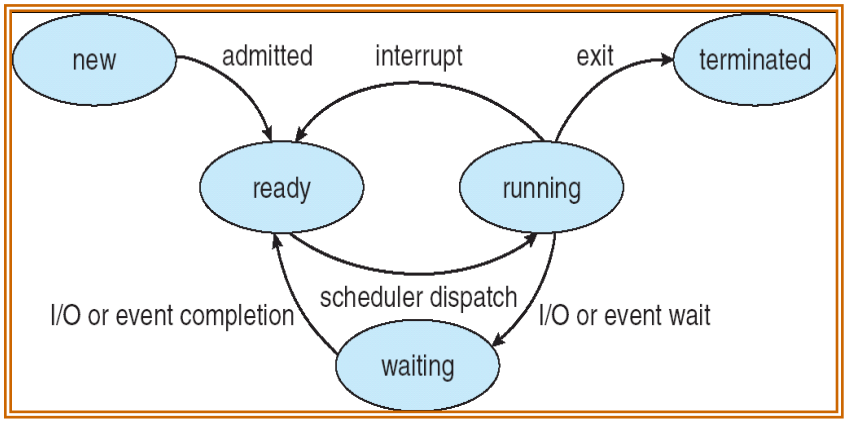
- **Multiprogramming is designed to achieve maximum use of resources, especially the CPU**
- **The CPU scheduler is the part of the OS that decides which process in the ready queue gets the CPU when it becomes free**
  - should aim for optimal use of the CPU
- **CPU is the most valuable resource in a computer, so we are talking about it**
  - However, the principles that we will see also apply to the scheduling of other resources (I / O units, etc.).
- **Must understand process behavior**
  - To make the right scheduling decision

The cycles of a process



- CPU and I / O cycles (bursts): the execution of a process consists of execution sequences on CPUs and I / O waits

When to invoke the scheduler



- **The scheduler must make its decision every time the executing process is interrupted, i.e.**
  - a process presents itself as new or finished
  - an executing process becomes stuck waiting
  - a process changes from running to ready
  - a process changes from waiting to ready
  - in conclusion, any event in a system causes an interruption of the CPU and the intervention of the scheduler, who will have to make a decision about which proc. or thread will have the CPU after
- **Preemption: we have preemption in the last two cases if we remove the CPU from a process that had it and can continue to use it**
- **In the first two cases, there is no preemption**
- **Several problems to resolve in the case of preemption**

Dispatcher

- **The code of the OS that gives control to the process chosen by the scheduler. It must be concerned with:**
  - change context
  - change to user mode
  - restart the chosen process
- **Dispatcher latency**
  - the time required to perform the duties of the dispatcher
  - it is often overlooked, it must be assumed that it is small compared to the length of a cycle

Scheduling criteria

- There will normally be several processes in the ready queue
- When CPU becomes available, which one to choose?
- The general idea is to make the choice in the interest of the efficient use of the machine
- But the latter can be judged according to different criteria ...

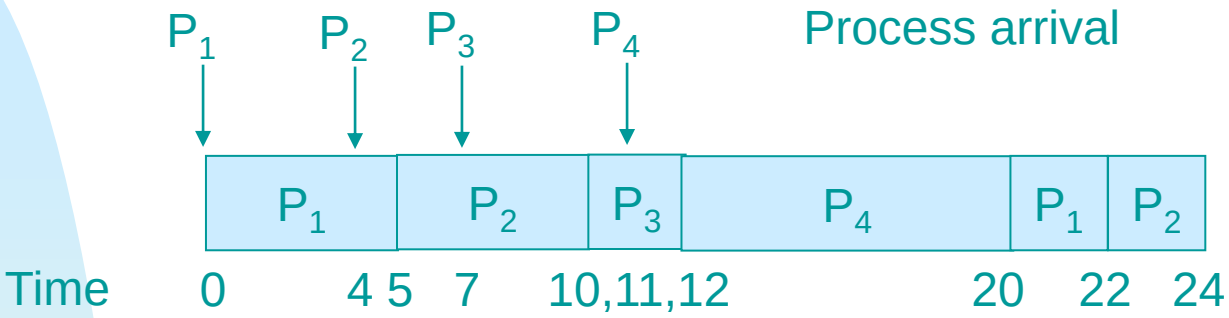
Scheduling criteria

- **Main reason for scheduling**
  - Percentage of use: keep CPUs and I / O modules busy
- **Time-sharing systems?**
  - Response time (for interactive systems): the time between a request and the response
- **Servers?**
  - Throughput: number of processes that complete in the unit of time
- **Batch processing systems?**
  - Turnaround time: the time taken by the process from its arrival to its end.
- **Loaded systems?**
  - Waiting time: waiting in ready queue (sum of all time spent in ready queue)

Scheduling criteria: maximize / minimize

- **To maximize**
  - CPU use: percentage of use
  - Throughput: number of processes that complete in the unit of time
- **To minimize**
  - Response time (for interactive systems): the time between a request and the response
  - Turnaround time: completion time minus arrival time
  - Waiting time: waiting in ready queue

Scheduling Criteria Example



- **CPU utilization:**
  - 100%
- **Throughput :**
  - 4/24
- **Turnaround time (P_3, P_2):**
  - P_3: 5
  - P_2: 20
- **Waiting time (P_2):**
  - P_2: 13
- **Response time (P_3, P_2):**
  - P_3: 3
  - P_2: 1

Now let's take a look at several scheduling methods and see how they behave against these criteria.

we will study specific cases

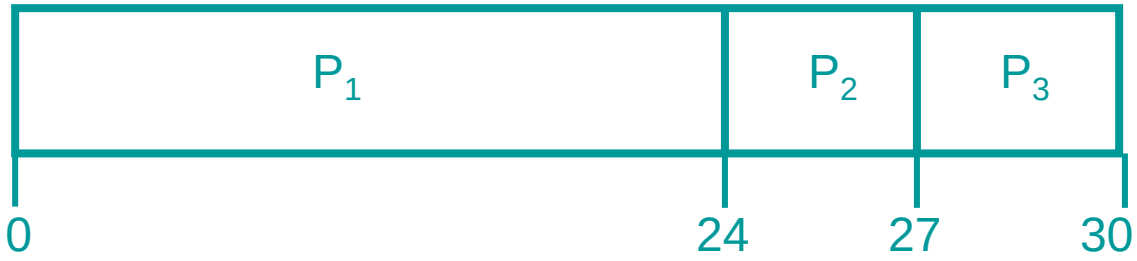
the study of the general case would require recourse to probabilistic or simulation techniques

First come, first served (First come, first serve, FCFS)

- Note, no preemption

Example: Process	Cycle time
P1	24
P2	3
P3	3

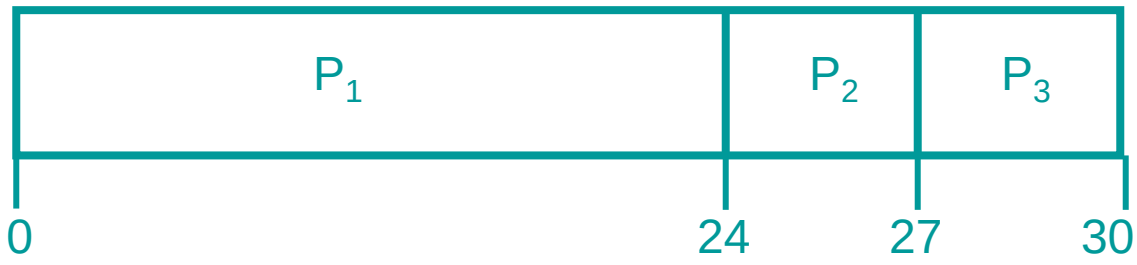
If the processes arrive at time 0 in the order: P1, P2, P3
The Gantt chart is:



Waiting time for P1 = 0; P2 = 24; P3 = 27
Average waiting time: (0 + 24 + 27) / 3 = 17

First come, first served

- **CPU utilisation = 100%**
- **Throughput = 3/30 = 0.1**
  - 3 processes completed in 30 units of time
- **Average rotation time: (24 + 27 + 30) / 3 = 27**



FCFS scheduling (continued)

If the same processes arrive at 0 but in order

$$P_2 , P_3 , P_1 .$$

The Gantt chart is:

- **Waiting time for $P_1 = 6$ $P_2 = 0$ $P_3 = 3$**
- **Average waiting time: (6 + 0 + 3) / 3 = 3**
- **Much better!**
- **So for this technique, the average wait time can vary greatly**
- ***Exercise: also calculate the average rotation time, throughput, etc.***

Take into account the arrival time!

- **If the processes arrive at different times, subtract the arrival times**
- **Exercise: repeat the calculations if:**
  - P2 arrives at time 0
  - P1 arrives at time 2
  - P3 arrives at time 5

Convoy Effect with the FCFS

- Consider a single CPU bound process and many I/O bound processes (fairly normal situation).
- The I/O bound processes wait for the CPU: I/O under utilized (*).
- The CPU bound process requests I/O: the other processes rapidly complete CPU bursts and go back to I/O: CPU under utilized.
- CPU bound process finishes I/O, and so do the other processes: back to *
- Solutions?

FCFS Scheduling Discussion

Is it simple and easy to program?

- Yes!

Can we find other benefits?

- It costs little time to make scheduling decision

Does it provide low waiting time?

- Not at all, can be quite awful

Does it provide good CPU utilization?

- No – the convoy effect

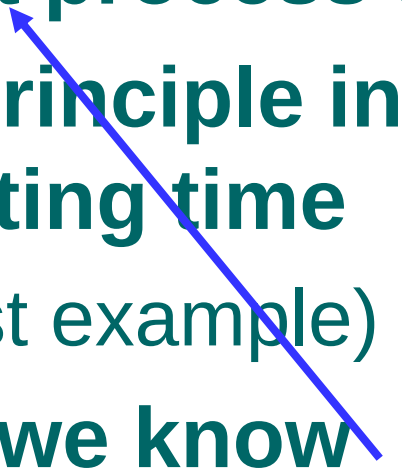
OK, forget about it.

Now, let's try to reduce the average waiting time

Idea: **It is better to execute ready processes with short CPU bursts**

Shorter First = Shortest Job First (SJF)

- **The shortest process starts first**
- **Optimal in principle in terms of average waiting time**
  - (see the last example)
- **But how do we know**



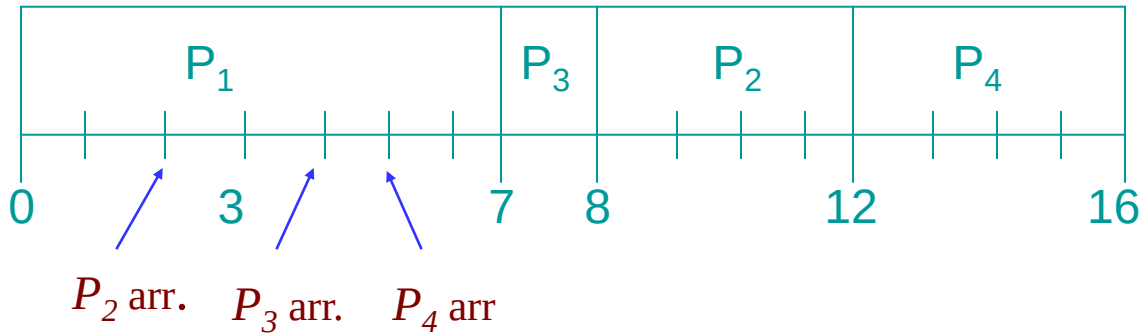
Shortest-Job-First (SJR) Scheduling

- **Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time**
- **Two schemes:**
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is also know as the Shortest-Remaining-Time-First (SRTF)
    - *Just call it preemptive SJF*
- **Preemptive SJF is optimal – gives minimum average waiting time for a given set of processes**
  - Moving a process with short CPU burst in front of a process with longer CPU burst reduces average waiting time

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

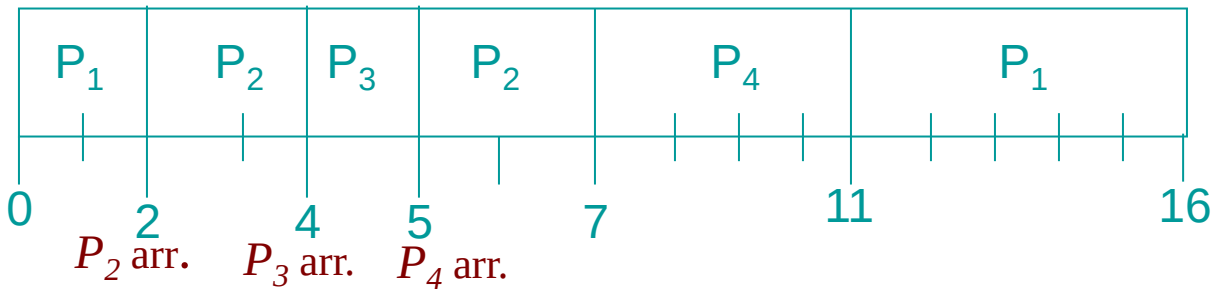


- Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Small technical detail with SJF

How do we know the length of the next CPU burst?

- If you know that, you probably know also tomorrow's stock market prices…
  - You will be rich and won't need to waste time in CSI3131 class
- So, we can only estimate it

Any idea how to estimate the length of the next CPU burst?

- Probably similar as the previous bursts from this process
- Makes sense to give more weight to the more recent bursts, not just straightforward averaging
  - Use exponential averaging

Exponential averaging with SJF

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \le \alpha \le 1$ = relative weight of recent vs past history
4. Define:

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\alpha t_{n-1} + (1-\alpha)^2 \alpha t_{n-2} + \ldots$$

$$+ (1-\alpha)^i \alpha t_{n-i} + \ldots + (1-\alpha)^n t_1$$

The shortest first SJF: review

- **Difficulty estimating the length in advance**
- **Long processes will suffer from *famine* when there is a constant supply of short processes**
- **Preemption is required for timeshared environments**
  - A long process can monopolize the CPU if it is the first to enter the system and it does not do I / O
- **There is an implicit assignment of priorities: preferences for shorter jobs**

SJF Discussion

Does it ensure low average waiting time?

- Yes, it was designed that way
  - As long as our burst-length predictions more-or-less work

Does it provide low response time?

- Not necessarily, if there is a steady stream of short CPU bursts, the longer bursts will not be scheduled
- This is called starvation
  - A process is blocked forever, always overtaken by other processes (well, or at least while the system is busy)

Let's see Priority Scheduling.

Priority Scheduling

- **A priority number (usually integer) is associated with each process**
  - On some systems (Windows), the higher number has higher priority
  - On others (Unix) , the smaller number has higher priority
- **The CPU is allocated to the process with the highest priority**
  - Can be preemptive or non-preemptive
  - but usually you want to preempt low-priority process when a high priority process becomes ready
- **Priority can be explicit**
  - Assigned by the sysadmin/programmer, often for political reasons
    - Professor jobs are of higher priority
  - But also for technical reasons
    - This device has to be serviced really fast, otherwise the vital data will be lost (real-time processing)
- **Priority can also be implicit (computed by the OS)**
  - SJF can be seen as priority scheduling where priority is the predicted next CPU burst time

Priority Scheduling Discussion

Good properties

- Professor jobs will be scheduled before student jobs
- Allows support of real-time processing

Bad properties

- Professor jobs will be scheduled before student jobs
  - OK, give me something else
- starvation – low priority processes may never execute

How to resolve the starvation problem?

- aging – keep increasing the priority of a process that has not been scheduled for a long time

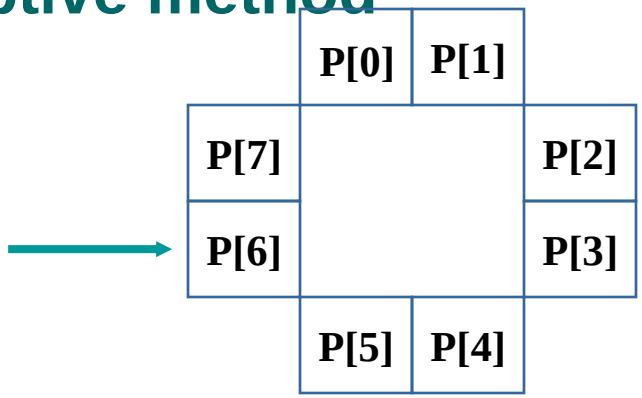
What to do with the processes of the same priority level?

- FCFS
- Might as well add preemption = Round Robin

Round-Robin (RR)

Most used in practice

- **Each process is allocated a quantum of time (e.g. 10-100 millisecs.) To run**
  - (book terminology: *time slice*)
- **If it runs for an integer quantum without other interruptions, it is interrupted by the timer and the CPU is given to another process**
- **The interrupted process becomes ready again (at the end of the queue)**
- **Preemptive method**



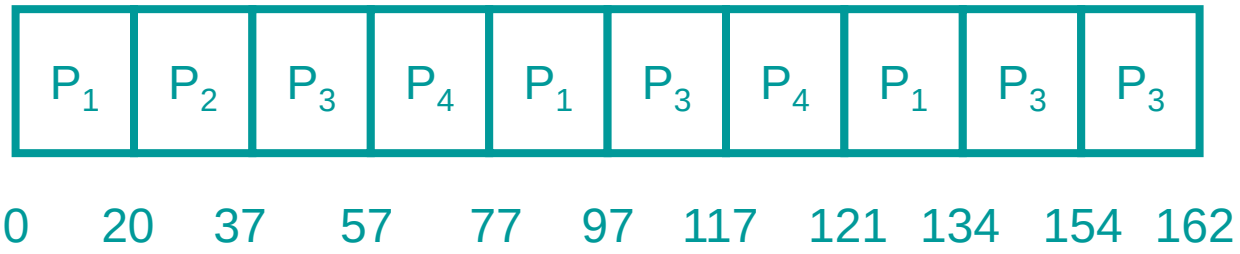
The ready queue is a circle (including RR)

Round-Robin performance

- If there is a process in the ready queue and the quantum is q, then each process receives 1 / n of the CPU time in units of max duration. q
- If q large $\Rightarrow$ FCFS
- If q small ... we will see

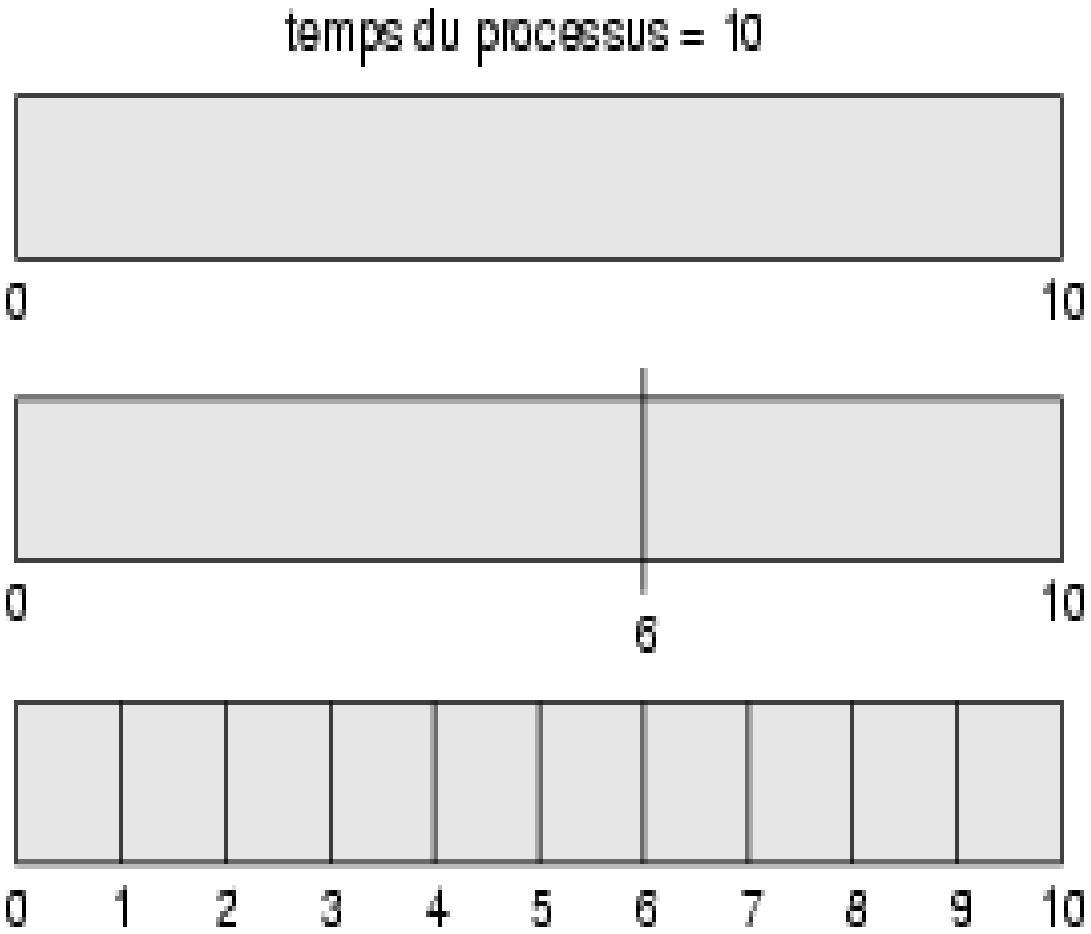
Example: RR Quantum = 20

Process	Cycle
P_1	53
P_2	17
P_3	68
P_4	24



- **Normally,**
  - higher turnaround time than SJF
  - but average waiting time better

A small quantum increases context switches (OS time)



tranches de temps	changement de contexte
12	0
6	1
1	9

Example to see the importance of a good choice of quantum (to be developed as an exercise)

- **Three cycles:**
  - A, B, C, all of 10
- **Try with:**
  - q = 1
  - q = 10
- **In this second case, the round-robin works as FCFS and the average rotation time is better**

Algorithms we have seen so far

- **First Come First Serve**
  - simple, little overhead, but poor properties
- **Shortest Job First**
  - needs to know CPU burst times
  - exponential averaging of the past
- **Priority Scheduling**
  - This is actually a class of algorithms
- **Round Robin**
  - FCFS with preemption

Scheduling exercises

Consider three processes P1, P2, P3

- **Burst times for P1: 14,12,17**
- **Burst times for P2: 2,2,2,3,2,2,2,3,2,2,2,3,2,2,2,3**
- **Burst times for P3: 6,3,8,2,1,3,4,1,2,9,7**
- **All three arrive at time 0, in order P1, P2, P3**
- **Each CPU burst is followed by an I/O operation taking 6 time units**
- **Let's simulate the scheduling algorithms**
  - FCFS
  - Round Robin (quantum=5)
  - Non-preemptive SJF or Preemptive SJF (your choice)
  - Round robin (quantum=5) with Priority scheduling, priorities are P2=P3>P1

Multilevel Queue

Idea: **Partition the ready queue into several queues, and handle each queue separately.**

Example:

- foreground queue (interactive processes)
- background (batch processes)

Each queue might have its own scheduling algorithm

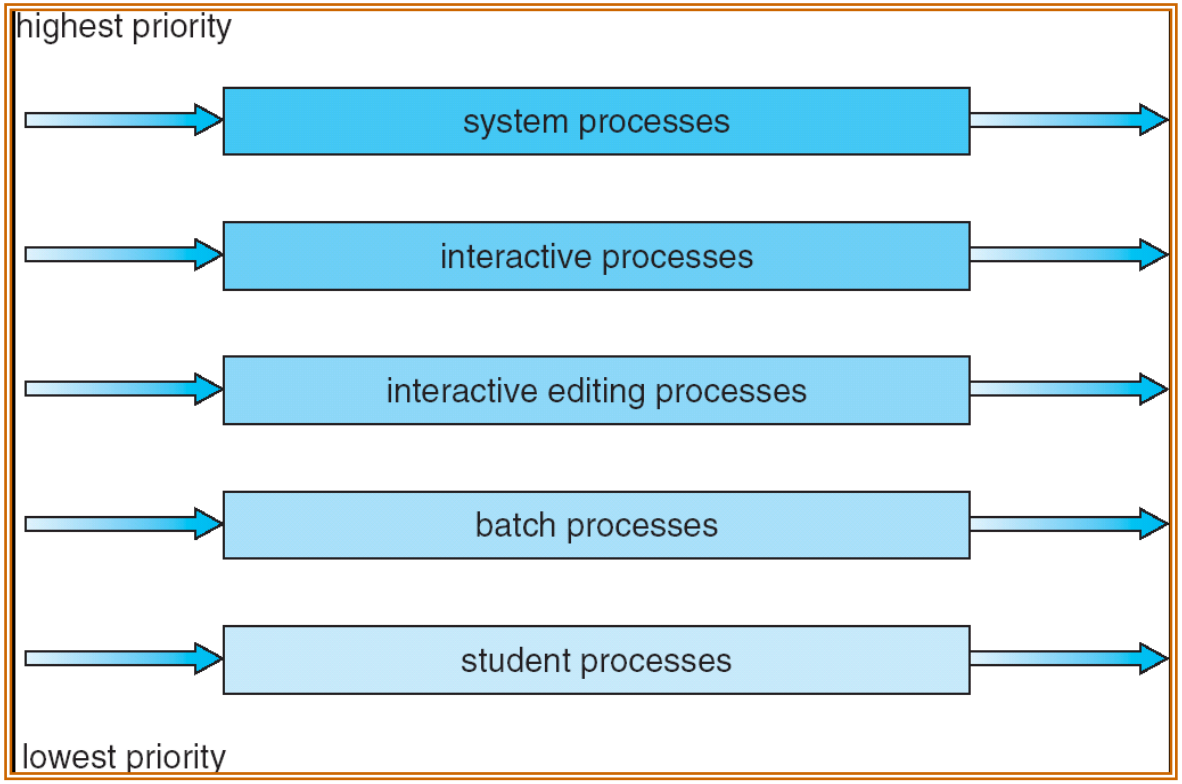
- foreground – RR (for low response time)
- background – FCFS (for simplicity and low context-switch overhead)

Multilevel Queue

How to schedule from among the queues?

- Fixed priority scheduling
  - i.e. the processes from foreground queue get the CPU, the background processes get the CPU only if the foreground queue is empty
  - Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes
  - i.e., 80% to foreground queue, 20% to background queue
  - not necessarily optimal

Multilevel Queue Scheduling



Multilevel Feedback Queue

Idea:

- **Use multilevel queues**
- **A process can move up and down queue hierarchy**
  - Why would a process move to a lower priority queue?
    - It is using too much CPU time
  - Why would a process move to a higher priority queue?
    - Has been starved of CPU for long time
    - A way to implement aging

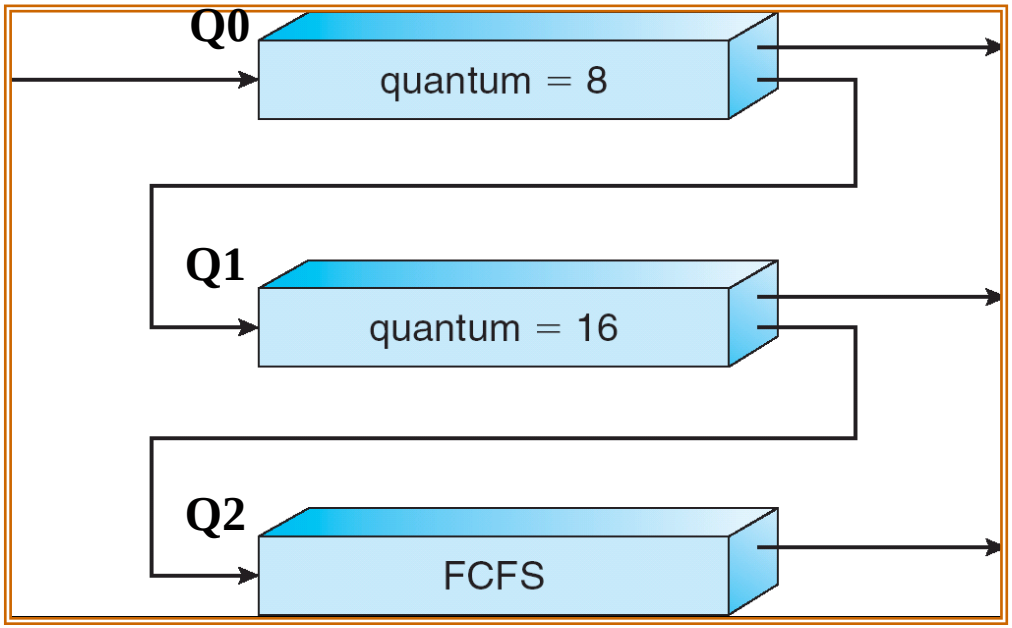
Multilevel Feedback Queue

- **Multilevel-feedback-queue scheduler defined by the following parameters:**
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service
- **This algorithm is the most general one**
  - It can be adapted to specific systems
  - But it is also the most complex algorithm

Example of Multilevel Feedback Queue

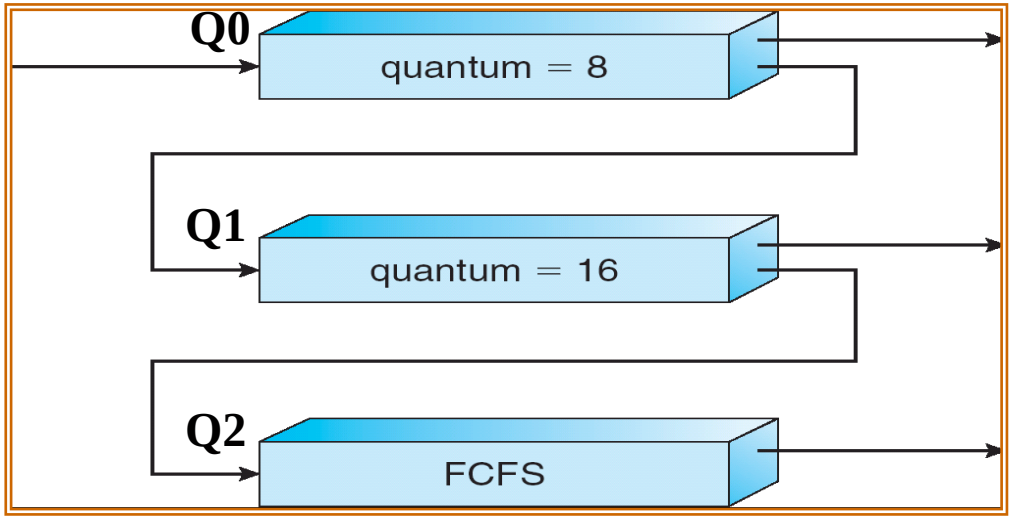
- **Scheduler selects processes in Q0 first (highest priority)**
  - If Q0 is empty, the processes from Q1 are selected.
  - If both Q0 and Q1 are empty, processes from Q2 are selected
- **If a process arrives in a higher priority queue when another from a lower priority queue is running, the running process will be preempted, to allow the arriving process to run.**

- **When a process exhausts its quantum in either Q0 or Q1, it is preempted and moved to the lower priority queue.**



Example of Multilevel Feedback Queue

- **Scheduling example**
  - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds.
  - If it does not finish in 8 milliseconds, job is preempted and moved to queue Q_1 and served again FCFS to receive another 16 additional milliseconds.
  - If it still does not complete, it is preempted and moved to queue Q_2.



Multilevel Feedback Queue Discussion

Exact properties depend on the parameters

Flexible enough to accommodate most requirements

The convoy example:

- One process with long CPU burst time
- Several I/O bound processes with short CPU burst time
- Even if the all processes start at the same level, the CPU-intensive process will be soon demoted to a low priority queue
- The I/O bound processes will remain at high priority and will be swiftly serviced, keeping the I/O devices busy

In practice...

- **The methods we have seen are all used in practice (except *pure* SJF which is impossible)**
- **Sophisticated OS provide the system manager with a library of methods, which it can choose and combine as needed after observing the behavior of the system.**

Also…

- **Our study of scheduling methods is theoretical, does not consider in detail all the problems that arise in CPU scheduling.**
- **Eg. CPU schedulers cannot give the CPU to a process for all the time it needs**
  - Because in practice, the CPU will often be interrupted by some external event before the end of its cycle.
- **Also, this study does not consider the execution times of the scheduler at all.**

Summary of scheduling algorithms

- **First come, first served (FCFS)**
  - simple, low system time (overhead), low quality
- **Shortest Job First (SJF)**
  - Must know processing times (not practical)
  - Must predict using the exponential mean of the past
- **Scheduling with priority**
  - A class of algorithms
- **Round-Robin**
  - FCFS with preemption
- **Multilevel Queues**
  - Possible to use different algorithms with each queue
- **Multilevel Feedback Queues**
  - Combines several techniques

Overview of advanced scheduling topics

- Scheduling with several identical CPUs
- Evaluation model

Multiple-Processor Scheduling

Good news:

- **With multiple CPUs, we can share the load**
- **We can also share the OS overhead**

Bad news:

- **We are expected to efficiently share the load**
- **Managing the OS structures gets more complicated**
- **The scheduling gets more complex**

Multiple-Processor Scheduling - Approaches

We assume *homogeneous processors*

- i.e. all processors have the same functionality

Still, the scheduler (and possibly the whole OS) might be run on a single CPU

- *Asymmetric multiprocessing* – only one CPU accesses the system data structures and makes scheduling decisions
  - alleviates the need for data sharing
  - but might become bottleneck
- *Symmetric multiprocessing (SMP)* – each CPU is self-scheduling
  - Either from a common ready queue, or from a private ready queue
  - Care must be taken when CPUs access common data structures
  - Virtually all modern operating systems support SMP including Windows XP, Solaris, Linux, Mac OS X.

SMP Scheduling Issues

- **Processor Affinity**
  - When process runs on a physical CPU, cache memory is updated with content of the process
  - If the process is moved to another CPU, benefits of caching is lost.
  - SMP systems try to keep processes running on the same physical CPU – know as processor affinity
- **Load Balancing**
  - Required when each CPU has its own Ready Queue
  - Push migration: a task is run periodically to redistribute load among CPUs (their ready queue)
  - Pull migration: An idle CPU (i.e. with an empty ready queue), will pull processes from other CPUs
  - Linux supports both techniques
  - Can counteract the benefits of processor affinity, since it moves processes from one CPU to another

Evaluation methods and comparison of algorithms

- Deterministic modeling
- Queuing models
- Simulation

Deterministic modeling

- **Essentially, what we have already done by studying the behavior of several algorithms on several examples**

Use of queuing theory

- Analytical method based on probability theory
- Simplified model: in particular, the OS times are ignored
- However, it makes estimates possible

Queuing theory: Little's formula

- **An important result:**
- **$n = \lambda \times W$**
  - n: average length of the queue
  - λ : process arrival rate in queue
  - W: average waiting time in the queue
- **Eg.**
  - λ if processes arrive 3 per sec.
  - **W** and they stay in the queue for 2 seconds
  - **n** the average queue length will be ???
- **Exercise: Solve also for λ and W**
- **Observe that for n to be stable, $\lambda \times$ W must be stable**
    - If n must remain 6 and λ goes up to 4, what must be W?

Simulation

- Build a *(simplified...)* model of the sequence of events in the OS
- Assign a duration of time to each event
- Assume a certain sequence of external events (e.g. arrival of process, etc.)
- Run the model for this sequence to get stats

Important points in this chapter

- **Ready Queue for CPU**
- **Scheduling criteria**
- **Scheduling algorithms**
  - FCFS: simple, not optimal
  - SJF: optimal, difficult implementation
    - exponential averaging
  - Priorities
  - Round-Robin: selection of the quantum
- **Algorithm evaluation, queuing theory,**
  - Little's formula


Thank You!
Ευχαριστώ
ขอบคุณ
Vielen Dank
Merci
Teşekkürler
DMnvwd
شكراً
متشكرم
Gracias
Grazie
Bedankt
Dankie
THANK YOU
Köszönettel
Obrigado!
شکریا
Díky
謝謝
GADDA GUEY
WAD MAHAD
SAN TAHAY
감사합니다
Urakoze